9 - Complex Data Structures

Charles A. Wuethrich

Bauhaus-University Weimar
CoGVis/MMC

January 13, 2020

Elementary Data Structures

- Every programming language offers to the programmer simple data types and simple constructs to organize simple data.

- Java allows the following data types and arrays:
  - `boolean`: a binary value, generally `true` or `false`,
  - `char`: a character, i.e. an 8 bit positive integer,
  - `byte`: an 8 bit integer,
  - `short`: a 16 bit integer,
  - `int`: a 32 bit integer,
  - `long`: a 64 bit integer,
  - `float`: a 32 bit floating point number,
  - `double`: a 64 bit floating point number.

  - `array[]`: a sequence of cells of the same type.
  - `String[]`: an array of characters.

Data Structures and Classes

- Simple data-types allow limited flexibility:
  For example, arrays must define their length when they are initialized.
- If you store data in arrays, when they are full, you have to define a bigger new array and copy everything.
- It would be nice to have the possibility of having "variable" size places where to store things...
  ... expanding and shrinking when needed.
- This is what *Linked Lists* are for!

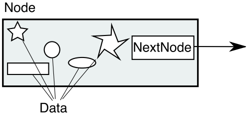
Linked Lists

- A *linked list* is a set of individual nodes, each of which is constituted by:
  - Object data (its variables, arrays and things....)
  - A pointer, or reference to the next element of the list.

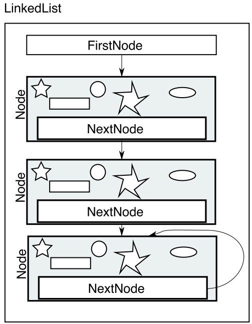
- The list itself contains only a reference to its first node
- The last node, by convention, points to itself (or to NULL).

- It is a bit like a locomotive and wagons attached to it: Each wagon is attached to the next one.
- Searching all the wagons means you start at the locomotive and search one after the other the wagons till the last one

Linked lists



(a) A node



(b) A list

Node implementations

- This is the corresponding code:

```
public class Node {
 NodeData data;
 Node nextnode;

 public Node(NodeData newdat) {
  data = newdat; nextnode = null;
 }
```

- Note how the node pointer is NULL as long as the node is not attached to a list.
- The constructor
  - Reserves memory for data and reference.
  - Initializes reference to NULL

Linked list implementation

- For the list class, first the node has to be filled with its data
- `thisdata` contains the node data
- and the node is made to point to itself (since it is the last)

```
Node newhead = new Node();
newhead.data = new NodeData(thisdata);
newhead.nextnode = newhead;
```

- Through `new` the variables of the data are filled by the constructor.

Linked list implementation

- Then comes the class `LinkedList` definition:

```
public class LinkedList {
 Node firstnode;

 public class NodeData {
  public NodeData(NodeData copy) {
  }
 }

 public class Node {
  public NodeData data;
  public Node nextnode;

  public Node(NodeData newdat) {
   data = newdat; nextnode = null;
  }

  public void initializehead
  (NodeData thisdata) {
   Node newhead;
   newhead = new Node(thisdata);
   newhead.nextnode = newhead;
  }
 }

 public LinkedList(Node newhead){
  // initialize LinkedList anew
  newhead.nextnode = newhead;
  firstnode=newhead;
 }
}
```

Linked Lists

- List contains only pointer to head
- If the first node does not exist, then the list points to NULL.

- Let us check if the list is empty:

```
public boolean CheckIfEmpty() {
 boolean empty = false;
 if(this.firstnode==null)
  empty = true;
 return empty;
}
```

Linked Lists

- Which methods belong to a list?
- We need to search nodes, add nodes, remove nodes.

- Add a node `tobeadded` at the head:

```
public void addNodeAtHead(Node tobeadded) {
 tobeadded.nextnode = this.firstnode;
 firstnode = tobeadded;
}
```

- We initialize the list by adding a new node and making the list point to this new head.

Linked Lists: add a Node

- Add a node `tobeadded` after the node `previousnode`:

```
public void addNodeAfter (Node previousnode,
 Node tobeadded ) {
 tobeadded.nextnode = previousnode.nextnode;
 previousnode.nextnode = tobeadded;
}
```

- Here we make `tobeadded` point to what `previousnode` was pointing to, and `previousnode` point to `tobeadded`

Linked Lists: remove a Node

- Remove a node `toberemoved`, and remember which its predecessor was:

```
public void removeNode
(Node previousnode, Node toberemoved ) {
 previousnode.nextnode =
 toberemoved.nextnode ;
 }
```

- Here we make `previousnode` point to what `toberemoved` was pointing to.

Linked Lists: search a Node

- We need to find a node in the list.
- We have to pass also who is its predecessor.

```
SearchNode(Node tobeseeked){
  NodeData data;
  Node=jumptonext;
  Node=previousnode;

  jumptonext=firstnode;
  previousnode=null;
  while((previousnode!=jumptonext)||
        (jumptonext.NodeData!=
           tobeseeked.NodeData)){
    previousnode = jumptonext;
    jumptonext=jumptonext.next;
    }
  if(previousnode==jumptonext){
    previousnode=null;
    jumptonext=null;
  }
  return(previousnode,jumptonext);
}
```

- The method returns NULL if the data was not found, `jumptonext` for the node found, and `previousnode` as its predecessor.

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

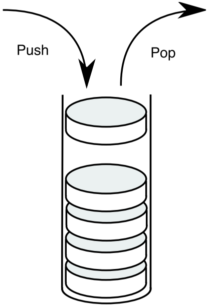
Stacks

- Sometimes, data has to be put into repository
- Like on my desk, it gets stacked
- Like in a tennis ball tube

- So one can recover it when necessary
- A tennis ball tube has a bottom
- One can insert the balls only from one side

Stacks

- Stacks are exactly like this: each data unit is like in a list.

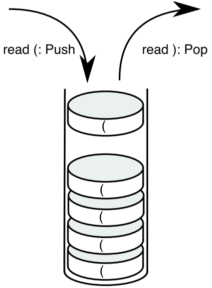


- The data is piled
- Only the top is accessible
- Two basic operations:
  - Pop
  - Push

Stacks

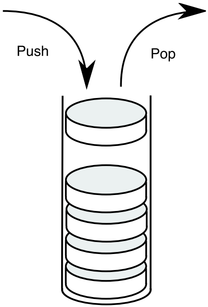
- Seems a little of an abstract data structure
- Let us think at a braces checker for an editor

- By pushing open braces in the stack when we find one
- . . . and popping them when we find a closed brace
- . . . we can count if the braces are correct!



Stacks Implementation

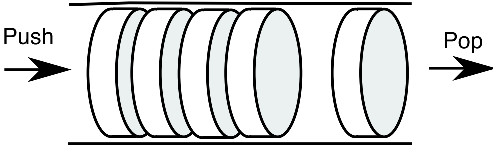
- Basically it is the same as Linked Lists:
  The node refers to the node below it.
- Only difference:
  - push inserts only at the head of the list
  - pop deletes first element in the list



- Of course, you need a method to check if the stack is empty.

Queues

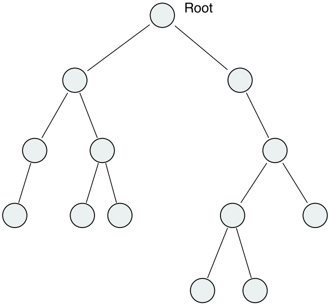
- Very similar to stack
- Only difference:
  - *push* on one side
  - *pop* at other side



- In the implementation the head needs to know also who the end of the queue is, not just the head

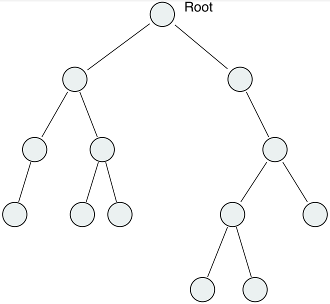
- Applications anyone?

Binary Trees



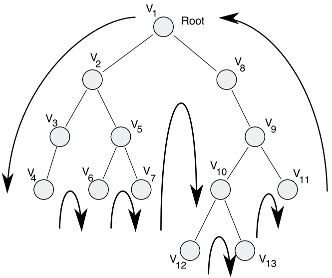
- A binary tree is a collection of nodes and links between them
- One node is a privileged node, called root and has no parents
- Each node has data stored in it, and
  - 0, 1 or 2 children
  - One parent node
- Nodes with no children are called *leaves*.

Implementing Binary Trees



- Tree Class points to the root of the tree
- Nodes are almost the same as in Linked Lists
- Only difference is there are two references:
  - One for the sibling node to the right (empty if none)
  - One for child node
- Of course, both can be empty
  → reference to `null` or to itself, depending on convention used

Traversing binary trees



- Traversing a tree means defining a path that touches all the nodes of the tree.
- Start at root node, move down and left to right, retrace up when at leaf or no sibling present
  - pre-order traversal: Visit first content of node, then children subtrees (left and right).
  - Post-order traversal: Visit first node children subtrees (left and right), then node itself.
  - In-order traversal: First visit left child subtree, the node itself, then right hand child.